Adaptive Layout Decomposition with Graph Embedding Neural Networks

Wei Li*, Jialu Xia*, Yuzhe Ma*, Jialu Li*, Yibo Lin†, Bei Yu*
*The Chinese University of Hong Kong †Peking University
{wli,byu}@cse.cuhk.edu.hk, yibolin@pku.edu.cn

***Abstract*— Multiple patterning lithography decomposition (MPLD) has been widely investigated, but so far there is no decomposer that dominates others in terms of both the optimality and the efficiency. This observation motivates us exploring how to adaptively select the most suitable MPLD strategy for a given layout graph, which is non-trivial and still an open problem. In this paper, we propose a layout decomposition framework based on graph convolutional networks to obtain the graph embeddings of the layout. The graph embeddings are used for graph library construction, decomposer selection and graph matching. Experimental results show that our graph embedding based framework can achieve optimal decompositions under negligible runtime overhead even comparing with fast but non-optimal heuristics.**

I. Introduction

The semiconductor industry nowadays is greatly challenged by extreme scaling which imposes severe issues on circuits manufacturing. Among various advanced lithography techniques, multiple patterning lithography (MPL) is one of the most practical solutions to enhance the manufacturability and has been widely adopted in industry [1].

The core problem of multiple patterning lithography is the layout decomposition which assigns features on a layout to separate masks for printability improvement and is also called multiple patterning lithography decomposition (MPLD). If two features located closer than minimum coloring distance are assigned to the same mask, a coloring conflict is introduced. Additionally, stitches can be inserted to assist conflict resolving, at a cost of potential yield loss though. Therefore, the objective of MPLD is to find a mask assignment for features such that the number of conflicts and stitches are minimized.

Due to the $\mathcal{NP}$-hardness of the general layout decomposition problem, a variety of decomposition approaches have been proposed to achieve high quality and efficiency. These approaches can be roughly categorized into three types: mathematical programming, graph-theoretical approaches and heuristic approaches. The mathematical programming approach formulates the problem into integer linear programming (ILP) [2]–[7], and its relaxations such as semi-definite programming (SDP) [5], linear programming (LP) [8] and discrete relaxation method [9]. Besides mathematical programming, graph-theoretical approaches resolve the problem with graph theories, e.g., the maximal independent set (MIS) [10], the shortest-path [11], [12], and the fixed-parameter tractable (FPT) [13] algorithms. Some heuristic approaches are also proposed in [5], [10], [14], [15], which are generally efficient but may have low quality. A recent work formulated MPLD into an exact cover problem and achieved high quality and efficiency with algorithm X [15]. Another extremely fast solution is based on graph matching [14], in which a coloring solution library for small graphs is constructed, and then graphs are colored efficiently by graph matching.

Although many decomposition algorithms have been developed, there is no conclusion that one decomposer is always better than another. ILP-based method ensures the optimality but suffers from runtime overhead for large layouts. Exact-cover (EC) based method demonstrates high efficiency for large layouts at a cost of marginal degradation on the solution quality. The graph matching based method shows good performance in both efficiency and quality for small graphs. But the library size of this method cannot be too large and only non-stitch graphs are supported, which is not applicable to large layouts or layouts with stitches. This observation motivates that it is worth exploring how to adaptively select the most suitable MPLD strategies for a given layout, which is non-trivial and still an open problem so far.

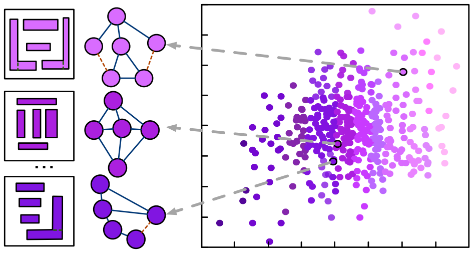


Fig. 1 An example of graph embeddings of layout graphs, where the graphs are transformed into vector space.

With successful deep learning applications in various fields by learning from historical data, we can naturally cast the problem into a classification task and leverage learning-based approaches. We need to investigate as much information of the graphs as possible and let our framework learn to adaptively utilize proper decomposition algorithms. However, graphs usually vary in terms of scale, making them hard to digest for learning models. Therefore, we need to obtain graph embedding under unified shape to represent the graph as shown in Fig. 1. Specifically, we use some techniques to generate the graph embedding such that the graph is transformed into a vector space in a lower but unified dimension with maximal representation capability and the powerful graph embedding helps us to adaptively select the best decomposer, where the best refers to the best solution quality at the lowest runtime.

Among different graph embedding methods, graph convolutional network (GCN) is widely used for irregular graph representations. In this paper, we adopt relational graph convolutional network (RGCN) to obtain graph embeddings, which is a variant of GCN to handle heterogeneous graphs. The graph embedding is then used as a representation to select ILP-based decomposer (optimal but slow) or EC-based decomposer (efficient but may not be optimal). Besides decomposer selection, the graph embedding helps us to avoid isomorphic graphs during library construction. After that, it is used for matching graphs efficiently in the library.

The main contributions are summarized as follows:

- We use an RGCN-based neural network which generates graph embedding of the layout graph.
- We design a graph library construction algorithm based on graph embeddings for small graphs excluding isomorphic ones.

978-1-7281-1085-1/20/$31.00 ©2020 IEEE

- We propose an adaptive workflow for efficient decomposer selection and graph matching using graph embeddings.
- We conduct experiments on widely used benchmarks and experimental results demonstrate that our framework can reduce the runtime by 87.7% while still preserving the optimality compared with optimal but slow ILP-based decomposer.

The rest of this paper is organized as follows. Section II introduces basic terminologies and evaluation metrics related to this work. Section III shows details of the GCN-based framework, including graph library construction and GCN model construction. Section IV covers experimental results and Section V concludes the paper.

II. Preliminaries

A. Multiple Patterning Lithography Decomposition (MPLD)

The input of MPLD is a layout specified by features in polygonal shapes, which is usually translated into an undirected heterogeneous graph $G = (V, E)$, where every node $v_i \in V$ corresponds to one complete or segmented feature in the layout and each edge $e_{ij} \in E$ represents a kind of relation between nodes. E is composed of conflict edge and stitch edge and is denoted by $E = \{E_c \cup E_s\}$, where E_s is the set of stitch edges and E_c is the set of conflict edges. The objective of MPLD problem is to minimize the weighted summation of the conflict number and the stitch number and can be formulated as:

$$\min \quad \sum c_{ij} + \alpha \sum s_{ij}, \tag{1}$$

where c_{ij}, s_{ij} are binary variables and represent conflict edge $e_{ij} \in E_c$ and stitch edge $e_{ij} \in E_s$ respectively. α is a parameter indicating the relative importance between the conflict cost and the stitch cost, which is usually set as 0.1. The values of c_{ij}, s_{ij} are determined by the colors of the two corresponding nodes v_i, v_j. Specifically, if v_i and v_j are assigned the same color, then $c_{ij} = 1$ ($s_{ij} = 0$). On the other hand, $c_{ij} = 0$ ($s_{ij} = 1$) when two nodes are assigned different colors.

B. Graph Isomorphism and Graph Matching

Intuitively, graph isomorphism problem is to decide whether two ordered graphs are identical after they are un-ordered. Graph matching is not only to decide whether they are identical or not, but also to give an order map of nodes if they are identical.

The formal definition of graph isomorphism and graph matching is stated as follows [16]: Given two graphs $G_1 = (V_1, E_1), G_2 = (V_2, E_2)$ with $|V_1| = |V_2|$, where V_1, V_2 and E_1, E_2 are corresponding node sets and edge sets, respectively. The object of graph matching is to find a node-to-node mapping $f : V_1 \rightarrow V_2$ such that $(u, v) \in E_1$ if and only if $(f(u), f(v)) \in E_2$. This is called an isomorphism if such a mapping f exists, and G_1 is said to be isomorphic to G_2.

In graph library construction, graph isomorphism is one of the most critical factors because $n! - 1$ isomorphic graphs of any valid graph will be re-collected in the library if no isomorphism-free techniques are used, where n is the number of nodes. Also, graph matching is inevitable when extracting the corresponding node coloring results stored in the graph library for the matched graph.

C. Graph Convolutional Network (GCN)

With the development and further study of Neural Network, GCN, as a branch of Neural Network, has shown promising results in many domains such as the graph embedding.

Generally speaking, GCN takes the graph as input and returns the node embeddings or graph embeddings. Usually, GCN is composed of two modules, aggregator and encoder, which exploit the neighborhood information and node attributes respectively. Specifically, for each node u in graph G, the aggregator is to aggregate neighbor v's representations $\boldsymbol{h}_v$ and obtain an intermediate representation $\hat{\boldsymbol{h}}_u$ such that the final graph embedding is able to contain graph structure information. Especially, one virtual additional edge is added to each node, i.e. a single self-connection whose weight is defined as 1 to guarantee that the latter layer's node representation can also be informed by the corresponding representation at the previous layer besides neighbors. Encoder is to multiply the aggregated representation $\hat{\boldsymbol{h}}_u$ with a learnable matrix followed with a non-linear activation function. GCN can be also explained in a message-passing way where the intermediate representations can be viewed as messages. The aggregation is the actual message-passing phase and each node passes its message to its neighbors along the edge. The encoder is served as the integration phase, in which each node integrates received the message and reduces it into its new message. Each message-pass and integration phase formulate one GCN layer. The representation after the final layer is called the node embedding of each node and the graph embedding by GCN is usually obtained by a summation or mean operation using node embeddings.

D. Problem Formulation

Given a set of layout graphs and two state-of-the-art decomposers, ILP-based decomposer and EC-based decomposer, our objective is to train an RGCN model to obtain the graph embeddings such that 1) the embeddings can be used to build a graph library for small graphs, recording the coloring solutions; 2) any new graph can find the best decomposer using its embedding; 3) any new small graph can find the coloring solution directly through graph matching with graphs in the library.

III. Algorithms

A. Overview

Our framework can be divided into two modules by whether the operation is needed in decomposition (online) or not (offline). The offline module is prepared before any decomposition, including graph library construction and RGCN model training.

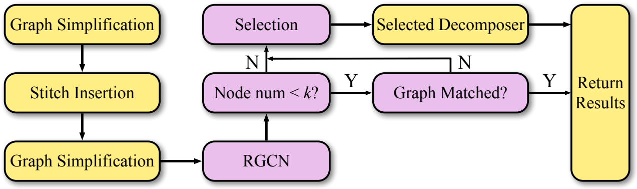


Fig. 2 The workflow of our framework. Purple blocks are executed in our framework while the yellow blocks are directly executed in OpenMPL [17].

Firstly, we train the RGCN model, then we use graph embeddings obtained by the trained RGCN model to build the isomorphism-free graph library. When the above offline steps are finished, we can execute layout decomposition following the workflow shown in Fig. 2. The input is transformed into a graph first and is simplified by several simplification techniques such as Independent Component Computation (ICC) [5], Hide Small Degrees [5], [14], Biconnected Component Analysis [3], [4]. Next, stitch candidates are inserted by pattern projection [5]. After stitch insertion, the simplified homogeneous graphs are transformed into heterogeneous graphs which contain both conflict and stitch edges and then these simplified heterogeneous graphs are fed into the RGCN model to obtain the graph embeddings. For a graph whose graph size is under the size constraint max_size, the corresponding graph embedding is first used to determine whether there is an isomorphism between the target graph and graphs in the library. If the isomorphic graph is found in the library, the corresponding node embeddings of two graphs are used to get the node-to-node mapping and directly return the final coloring result by the mapping in the library. If no isomorphic graph is found or the graph size is larger than max_size, the graph embedding is followed by a fully connected layer for decomposer selection and the graph is then decomposed by the selected decomposer. After all graphs are decomposed, a color recovery process is executed to get the final layout decomposition results.

B. Graph Embedding Neural Network

Graph embedding neural network is one of the most critical parts in our framework since the graph embedding obtained by the neural network is the basis for every module. Considering that the simplified graph is heterogeneous, which contains both conflict and stitch edges, we applied Relational Graph Convolutional Networks (RGCN) similar to [18] to obtain the graph embedding. The process for graph embedding is shown in Fig. 3. The original layout is transformed into multiple heterogeneous graphs by graph simplification and stitch insertion. Those simplified graphs are the input of the model and the model is composed of two neural network layers. For each node v_i in a graph $G = \{V, E\}, E = \{E_c, E_s\}$, the node representation $\boldsymbol{u}_i^{(l+1)} \in \mathbb{R}^{D^{(l+1)}}$ at the $(l+1)_{th}$ layer of the neural network can be calculated by the following formula:

$$\boldsymbol{u}_i^{(l+1)} = ReLU\left(\sum_{e\in E}\sum_{j\in N_i^e} \boldsymbol{W}_e^{(l)}\boldsymbol{u}_j^{(l)} + \boldsymbol{u}_i^{(l)}\right), \tag{2}$$

where $D^{(l)}$ is the dimension of node representation at the l_{th} layer, $\boldsymbol{W}_e^{(l)} \in \mathbb{R}^{D^{(l+1)}\times D^{(l)}}$ is a learnable weight matrix of edge type $e \in E$ and N_i^e denotes the set of neighbor nodes of node v_i connected by e. Intuitively, RGCN specified in Equation (2) works like the classical GCN, as both neural network layers contain two phases, aggregation and encoding. The difference is that edges in GCN share the same learnable weight in each layer on the encoding phase while only edges in the same edge type share the weight matrix for RGCN, which means that the message integration for different kinds of edges is independent. One central issue resulted from the different weight matrixes strategy is that the number of parameters rapidly grows with the number of the edge categories in the graph. Also, this kind of strategy can easily lead to overfitting due to a large number of parameters. The issue is solved by regularization of weight and we adopt a basis decomposition [18], in which each weight matrix $\boldsymbol{W}_e^{(l)}$ is a linear combination of basis transformations $\boldsymbol{V}^{(l)}$ and defined by:

$$\boldsymbol{W}_e^{(l)} = \sum_{b=1}^{B} \delta_{rb}^{(l)} \boldsymbol{V}_b^{(l)}, \tag{3}$$

where $\boldsymbol{V}_b^{(l)} \in \mathbb{R}^{D^{(l+1)}\times D^{(l)}}$ is one of the multiple basis transformations and $\delta_{rb}^{(l)}$ is the learnable coefficient.

In our implementation, we also adopt widely-used ReLU as the activation function, the input feature of node v_i is defined as:

$$\boldsymbol{u}_i^{(0)} = \sum_{j\in N_i} I_{\{e_{i,j}\in E_c\}} + \alpha I_{\{e_{i,j}\in E_s\}}, \tag{4}$$

where $I_{\{\cdot\}}$ is an indicator function and $\alpha = -0.1$ is a user-defined parameter following the general stitch cost. After obtaining the node embeddings by RGCN model, we calculate the graph embedding by the summation of the node embeddings considering that graph complexity influences the decomposition quality of EC-based decomposer, i.e., $\boldsymbol{h} = \sum_{i\in V} \boldsymbol{u}_i^{(out)}$, where $\boldsymbol{u}_i^{(out)}$ is the node embedding of node v_i.

Algorithm 1 Graph Library Construction

```
Require: $max\_size$ → Maximal graph size.
Ensure: $L$ → The isomorphism-free library of valid graphs;
 1: $L \leftarrow \{\}$;
 2: $S_p \leftarrow$ Generate graphs following method in [19];
 3: $S_p \leftarrow$ Remove invalid Graphs in $S_p$;
 4: $S \leftarrow$ Enumerate graphs containing stitches from graphs in $S_p$;
 5: for $G \in S$ do
 6:     if $G$ satisfies layout graph rules then
 7:         $\boldsymbol{h} \leftarrow$ normalize(RGCN($G$));
 8:         $\mathcal{L}_h \leftarrow$ Extract graph embeddings stored in the library;
 9:         if $\max(\mathcal{L}_h \times \boldsymbol{h}) < 1$ then
10:             Decompose $G$ with ILP-based decomposer;
11:             Insert $G$ into $L$;
12:         end if
13:     end if
14: end for
```

C. Graph Library Construction

Generally speaking, it is possible to enumerate all the valid graphs under the size constraint such that we can build up a graph library to accelerate decomposition by simply matching the graph with graphs in the library and collect the coloring information stored in the library.

Previous work [14] constructed a graph library that contains all homogeneous graphs (23 in total) with node number less than seven following the algorithm described in [19], [20]. However, the graph in the previous library does not contain stitch edge, which means that one heuristic stitch insertion and coloring method should be used if the no-stitch graph is not colorable. Therefore, we propose an isomorphism-free heterogeneous graph library construction algorithm that contains all the possible graphs with both stitch edges and conflict edges.

We first define the target heterogeneous graph as $G = \{V, E\}$, where V, E are the node set and the edge set respectively. Furthermore, we define a corresponding parent graph $G^p = \{V^p, E^p\}$, which is the no-stitch form of G by merging nodes connected by stitch edges. Different from the general 2-connected graph described in [19], the graph transformed by circuit layout has some specific rules, especially after stitch insertion. The rules are stated as follows:

- G^p is a 3-connected graph instead of 2-connected.
- The degree of each node in G is at least two.
- One node pair $\{u, v\}$ cannot be connected if u, v are in the stitch relation. Stitch relation of two nodes means that there is a path connecting them and only go through stitch edges with length larger than one.
- The neighbors connected by conflict edge cannot be totally the same for two nodes in stitch relation.

The pseudocode of our library construction algorithm is illustrated in Algorithm 1. Firstly, we enumerate G^p by the method in [19] (line 2), which generates isomorphism-free 2-connected graph set and removes all graphs which are not 3-connected (line 3). Then for each G^p, we enumerate valid G which satisfies the size constraint and all the rules above by splitting nodes in G^p and insert stitch edges (lines 4–6). Note that there may be multiple isomorphic graphs in the enumeration of G such that we use graph embedding to avoid isomorphism. Specifically, every time when the enumerated G is going to put into the library, G will be fed into the RGCN model (line 7) and obtain a corresponding normalized graph embedding $\boldsymbol{h} \in \mathbb{R}^D$, where D is the dimension of the graph embedding. Then

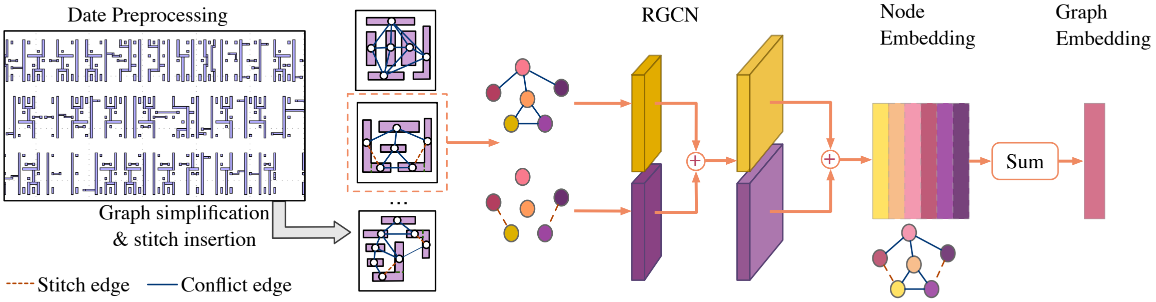


Fig. 3 Overview of the process for graph embedding

the normalized graph embeddings $\mathcal{L}_h \in \mathbb{R}^{k\times D}$ stored in the library are extracted (line 8) and a vector-matrix multiplication is performed i.e., $\boldsymbol{m} \in \mathbb{R}^k = \mathcal{L}_h \times \boldsymbol{h}$, where k is the number of graphs stored in the library temporarily. Then whether there is an isomorphic graph in the library or not is determined by checking the maximum element in $\boldsymbol{m}$ (line 9) because two unit vectors are equal if and only if their product is 1. The idea is based on the fact that a GCN-based model is insensitive to the node order, which means that the graph embeddings of all isomorphic graphs by a GCN-based model are totally the same. After isomorphism determination, G won't be inserted into the library if there is an isomorphic graph. Otherwise, G will be decomposed by ILP-based decomposer for optimal solution (line 10), then graph G with its optimal coloring result, corresponding graph embedding and node embeddings will be stored in the library (line 11).

D. Graph Matching and Decomposer Selection

1) Graph Matching

When the graph embedding is obtained by our model and the graph size is under the limitation, we directly match the graph with graphs in the library. We use the obtained graph embedding to find isomorphic graphs in the library, then we use the corresponding node embeddings to find the node-to-node mapping and return the solution directly.

To illustrate the process clearly, we provide a simple example and explain the details step by step. The graph library $\mathcal{L}$ in this example is composed of three graphs, in which each graph has four nodes and the dimension of graph embedding is two. The library stores all information of graphs needed by our framework including its node embeddings $\mathcal{L}_u \in \mathbb{R}^{3\times4\times2}$, graph embeddings $\mathcal{L}_h \in \mathbb{R}^{3\times2}$ and optimal solutions $\mathcal{L}_s \in \mathbb{R}^{4\times3}$:

$$\mathcal{L}_u = \begin{bmatrix} 0.4 & 0.4 \\ 0.3 & -1.0 \\ 0.1 & 0.6 \\ -0.2 & 0.8 \end{bmatrix} \quad \mathcal{L}_h = \begin{bmatrix} 0.6 & 0.8 \\ 0.6 & -0.8 \\ 1 & 0 \end{bmatrix} \quad \mathcal{L}_s = \begin{bmatrix} 1 & 1 & 0 \\ 2 & 1 & 1 \\ 0 & 2 & 0 \\ 1 & 0 & 2 \end{bmatrix}$$

Different colors represent different graphs in the library. Take a target graph G with four nodes for example, we use RGCN model to obtain the corresponding node embedding $\boldsymbol{u} \in \mathbb{R}^{4\times2}$ and graph embedding $\boldsymbol{h} \in \mathbb{R}^2$, where $\boldsymbol{h} = \sum_i \boldsymbol{u}_i$:

$$\boldsymbol{u} = \begin{bmatrix} 0.3 & -1.0 \\ -0.2 & 0.8 \\ 0.4 & 0.4 \\ 0.1 & 0.6 \end{bmatrix}, \boldsymbol{h} = \begin{bmatrix} 0.6 \\ 0.8 \end{bmatrix}. \tag{5}$$

We first multiply the graph embedding $\boldsymbol{h}$ with graph embeddings $\mathcal{L}_h$ in the library i.e., $\boldsymbol{m} \in \mathbb{R}^3 = \mathcal{L}_h \times \boldsymbol{h}$:

$$\boldsymbol{m} = \underbrace{\begin{bmatrix} 0.6 & 0.8 \\ 0.6 & -0.8 \\ 1 & 0 \end{bmatrix}}_{\mathcal{L}_h} \times \underbrace{\begin{bmatrix} 0.6 \\ 0.8 \end{bmatrix}}_{\boldsymbol{h}} = \begin{bmatrix} 1 \\ -0.28 \\ 0.6 \end{bmatrix}$$

Then the matched graph index i in the library is defined by:

$$i = \begin{cases} \arg\max(\boldsymbol{m}), & \text{if} \max(\boldsymbol{m}) = 1; \\ -1, & \text{otherwise}, \end{cases} \tag{6}$$

where -1 means there is no isomorphic graph matched in the library such that the graph matching process is terminated and redirected to decomposer selection, otherwise, the i_{th} node embedding in the library is extracted and compared with the target graph's node embedding to get the final node-to-node mapping. This comparison method is also based on the node order insensitivity of the GCN-based model, if the input feature doesn't contain any information related to the node order such as a one-hot vector of the node order, the final graph embedding is then order-invariant because the message passing process is only related to the neighbors instead of the node order. In this example, $\boldsymbol{m}[0] = 1$ such that $i = 0$, which means that the first node embedding $\mathcal{L}_u[0]$ is used to compare with $\boldsymbol{u}$.

The node-to-node mapping f is executed by comparing two node embeddings and formulated by:

$$f(j) = k, \text{ if } \boldsymbol{u}[j] = \mathcal{L}_u[i][k] \text{ for } j, k \text{ in } \{0, \dots, |G| - 1\}, \tag{7}$$

where $|G|$ means the number of nodes in the graph. In this example, $|G|$ is exactly 4 and f is then defined by: $f(\{0, 1, 2, 3\}) = \{1, 3, 0, 2\}$.

After f is found, the solution $\boldsymbol{s}$ can be matched quickly by:

$$\boldsymbol{s}[j] = \mathcal{L}_s[f(j)][i], \text{ for } j \text{ in } \{0, \dots, |G| - 1\}, \tag{8}$$

so the final solution of G in this example is mapped as $[2, 1, 1, 0]$.

2) MPL Decomposer selection

When the size is larger than the size limitation or no mapping is found in the library, the graph embedding is used to select the decomposer. Therefore, the decomposer selector can be regarded as a 2-class classifier and simply modeled by a summation of one trainable weight matrix $\boldsymbol{W}_s \in \mathbb{R}^{2\times D}$ and a bias vector $\boldsymbol{b}_s \in \mathbb{R}^2$ combined with $\arg\max$ function, which can be formulated as:

$$y = \arg\max(\boldsymbol{W}_s\boldsymbol{h} + \boldsymbol{b}_s), \tag{9}$$

where $\boldsymbol{h} \in \mathbb{R}^D$ is the graph embedding obtained by RGCN model with dimension D. The final decomposition result is then generated by the selected decomposer.

TABLE I Decomposition Cost Comparison

Circuit	ILP				SDP				EC				RGCN			
	st#	cn#	cost	time (s)	st#	cn#	cost	time (s)	st#	cn#	cost	time (s)	st#	cn#	cost	time (s)
C432	4	0	0.4	0.486	4	0	0.4	0.016	4	0	0.4	0.005	4	0	0.4	0.007
C499	0	0	0	0.063	0	0	0	0.018	0	0	0	0.011	0	0	0	0.015
C880	7	0	0.7	0.135	7	0	0.7	0.021	7	0	0.7	0.010	7	0	0.7	0.014
C1355	3	0	0.3	0.121	3	0	0.3	0.024	3	0	0.3	0.011	3	0	0.3	0.015
C1908	1	0	0.1	0.129	1	0	0.1	0.024	1	0	0.1	0.017	1	0	0.1	0.031
C2670	6	0	0.6	0.158	6	0	0.6	0.044	6	0	0.6	0.035	6	0	0.6	0.046
C3540	8	1	1.8	0.248	8	1	1.8	0.086	8	1	1.8	0.032	8	1	1.8	0.038
C5315	9	0	0.9	0.226	9	0	0.9	0.106	9	0	0.9	0.039	9	0	0.9	0.049
C6288	205	1	21.5	5.569	203	4	24.3	0.648	203	5	25.3	0.151	205	1	21.5	0.154
C7552	21	1	3.1	0.872	21	1	3.1	0.157	21	1	3.1	0.071	21	1	3.1	0.111
S1488	2	0	0.2	0.147	2	0	0.2	0.031	2	0	0.2	0.013	2	0	0.2	0.016
S38417	54	19	24.4	7.883	48	25	29.8	1.686	54	19	24.4	0.329	54	19	24.4	0.729
S35932	40	44	48	13.692	24	60	62.4	5.130	46	44	48.6	0.868	40	44	48	1.856
S38584	117	36	47.7	13.494	108	46	56.8	4.804	116	37	48.6	0.923	117	36	47.7	1.840
S15850	97	34	43.7	11.380	85	46	54.5	4.320	100	34	44	0.864	97	34	43.7	1.792
average			12.893	3.640			15.727	1.141			13.267	0.225			12.893	0.448
ratio			1.000	1.000			1.220	0.313			1.029	0.062			1.000	0.123

IV. Experimental Results

The experiments are performed on the scaled-down and modified ISCAS benchmarks, which are widely used in previous works [5], [14], [15]. The framework is mainly implemented in Python with PyTorch [21] and DGL [22] and integrated into the open-source layout decomposition framework OpenMPL [17]. Fig. 2 specifies the detailed task execution platform of the workflow. It should be noted that our graph embedding as well as the whole framework is very general that they can be naturally extended to other decomposition tasks under different lithography constraints. We follow the same settings in [5], [14], [15] on the minimum color space, where the first ten cases are set to 120 nm and the last five cases are set to 100nm. The cost of stitch is set to 0.1 such that the decomposition cost is calculated by $cn\# + 0.1st\#$, mask number is set to 3 and the graph simplification level in OpenMPL is 3. In the training phase of our model, we concatenate the model with MPL decomposer selector such that the cross-entropy loss function can be adopted. The label of each simplified graph for training is set as 0 (ILP) if the cost by ILP-based decomposer is smaller than EC-based decomposer and 1 (EC) for other cases. The RGCN model contains two layers whose output dimensions are 32, 64 respectively such that the dimension of graph embedding is 64. The training strategy follows the idea of K-fold cross-validation, specifically, each time one of the 15 layouts in the benchmark is used as the validation set and the other 14 layouts are put together to form a training set. Therefore, there are 15 trained models for 15 layouts following the same model configurations. The layout is first preprocessed by graph simplification and stitch insertion such that the dataset is composed of multiple graphs. Considering that our dataset is significantly unbalanced since EC-based decomposer is optimal and also the fastest in most cases, we set the training epoch to 1 and use a weighted random sampling strategy with weight ratio 300:1 to avoid overfitting. In the evaluation phase, the simplified graphs of the target layout are fed as a batch to the RGCN model for efficient inference. All the experiments are conducted on an Intel Core 2.9 GHz Linux machine with one NVIDIA TITAN Xp GPU.

In the first experiment, we compare the effectiveness of our proposed RGCN model with conventional GCN model. The classical GCN model only supports homogeneous graphs while there are two kinds of edges in this task. Therefore, we slightly modify the message passing function by multiplying the edge weight α_e for different edge types:

$$\boldsymbol{u}_i^{(l+1)} = ReLU\left(\sum_{e \in E} \sum_{j \in N_i^e} \alpha_e \boldsymbol{W}^{(l)} \boldsymbol{u}_j^{(l)} + \boldsymbol{u}_i^{(l)}\right), \tag{10}$$

where α_e is 1 for conflict edge and -0.1 for stitch edge following the weighted cost setting, the negative sign is due to the fact the stitch edge and conflict edge play different roles in decomposition: nodes connected by a conflict edge are assigned different colors while stitch edge indicates same color. The result is illustrated by the confusion matrix shown in Fig. 4, where each row contains the number of graphs selected to be decomposed by the corresponding decomposer while each column contains the number of graphs labeled by the corresponding decomposer. For example, the element (0,0) in the confusion matrix indicates the number of graphs which is labeled as positive (ILP) and also selected to be decomposed by ILP-based decomposer. In the experiment, we use two more metrics, recall and F1 score. Recall is used to measure the proportion of ILP-labeled graphs that are correctly identified and influences the decomposition quality directly. F1-score is a general metric for the model's accuracy. According to Fig. 4, we can see that the F1-score of our model is more than 2× of that in conventional GCN, which demonstrates the powerful representation capability of our model compared with conventional GCN. Another important point is that our model classifies all the graphs labeled as positive correctly such that our recall achieves 100% while conventional GCN only classifies 15.4% correctly.

		Label	
		ILP	EC
Predicted	ILP	13	682
	EC	0	5900
Recall		**100.0%**	
F1-score		**0.0367**	

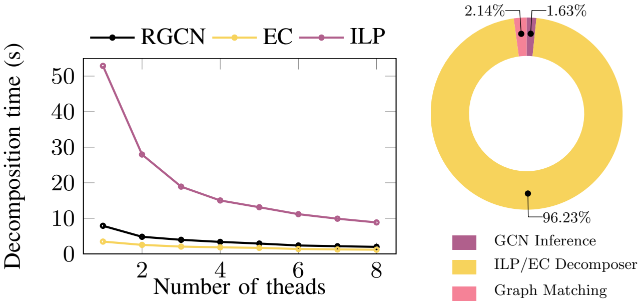
(a) Proposed RGCN

		Label	
		ILP	EC
Predicted	ILP	2	244
	EC	11	6338
Recall		15.4%	
F1-score		0.0154	

(b) Conventional GCN

Fig. 4 F1 score comparison of (a) RGCN (b) GCN.

In the second experiment, we compare our results with state-of-the-art decomposers. All the decomposers are implemented and measured in OpenMPL under one thread such that we can keep the preprocess procedure the same and compare the results without potential bias due to different simplification method or stitch insertion techniques. The results can be found in TABLE I, where the column "time (s)" is the decomposition runtime regardless of graph



(a)

(b)

Fig. 5 (a) Scalability analysis on different threads; (b) Runtime breakdown of our framework.

simplification and stitch insertion for better comparison. As expected, there is no one decomposer which can dominate others. EC-based decomposer outperforms others on runtime while causing some additional costs. ILP-based decomposer obtains the optimal results while the runtime is significantly worse than others. SDP-based decomposer shows a runtime improvement compared with ILP-based decomposer but cannot compete with EC-based decomposer on both runtime and quality. Our RGCN-based framework also obtains the optimal results in all cases since the selector selects all graphs labeled as ILP correctly and such avoid optimality loss. The average runtime is reduced to 12.3% compared with ILP-based decomposer because of the efficient graph matching technique and EC-based decomposer which is selected as the decomposer in most cases.

In the third experiment, we analyze the runtime of our framework. We first measure the runtime of different decomposers under different threads. The result is shown in Fig. 5(a), where each method is executed under different threads and the decomposition runtime is the total runtime of 15 layouts in the benchmark. According to the figure, EC-based decomposer demonstrates impressive efficiency under all threads. Despite that ILP-based decomposer can achieve optimal results, the runtime is substantially longer than other decomposers. The proposed RGCN model can achieve comparable decomposition quality to ILP-based decomposer, with a similar runtime to EC-based decomposer. We further compare the runtime distribution in our framework. The decomposition runtime of our framework is mainly composed of three parts: RGCN inference, Graph matching and the decomposition runtime by the selected decomposer. Note that the decomposer selection is counted in the RGCN inference since the 2-class classifier is integrated into the RGCN model for fast inference. Fig. 5(b) shows the result, where the metric is the total runtime of 15 layouts as before. From the figure, we can clearly see that the decomposition runtime by the selected decomposer is the major bottleneck and occupies 96.23% of the total runtime, and the RGCN inference runtime is even slightly faster than graph matching due to the batched graph technique we used in the evaluation phase. The result indicates that the RGCN inference and graph matching runtime of our framework are actually trivial such that our method has strong scalability and can be applied to select other more efficient decomposers in the future.

V. Conclusion

In this paper, we use an RGCN model to obtain graph embeddings, which are used to build the isomorphism-free graph library, match graphs in the library and adaptively select decomposer. The results show that the obtained graph embeddings have powerful representation capability and demonstrate an excellent balance between decomposition quality and efficiency. The future work can be the construction of more complicated libraries for efficient matching.

VI. Acknowledgment

This work is partially supported by Cadence Design Systems, NVIDIA, and The Research Grants Council of Hong Kong SAR (No. CUHK24209017).

References

[1] D. Z. Pan, B. Yu, and J.-R. Gao, “Design for manufacturing with emerging nanolithography,” *IEEE TCAD*, vol. 32, no. 10, pp. 1453–1472, 2013.

[2] Y. Xu and C. Chu, “GREMA: graph reduction based efficient mask assignment for double patterning technology,” in *Proc. ICCAD*, 2009, pp. 601–606.

[3] A. B. Kahng, C.-H. Park, X. Xu, and H. Yao, “Layout decomposition approaches for double patterning lithography,” *IEEE TCAD*, vol. 29, pp. 939–952, June 2010.

[4] K. Yuan, J.-S. Yang, and D. Z. Pan, “Double patterning layout decomposition for simultaneous conflict and stitch minimization,” *IEEE TCAD*, vol. 29, no. 2, pp. 185–196, Feb. 2010.

[5] B. Yu, K. Yuan, D. Ding, and D. Z. Pan, “Layout decomposition for triple patterning lithography,” *IEEE TCAD*, vol. 34, no. 3, pp. 433–446, March 2015.

[6] B. Yu, Y.-H. Lin, G. Luk-Pat, D. Ding, K. Lucas, and D. Z. Pan, “A high-performance triple patterning layout decomposer with balanced density,” in *Proc. ICCAD*, 2013, pp. 163–169.

[7] B. Yu, S. Roy, J.-R. Gao, and D. Z. Pan, “Triple patterning lithography layout decomposition using end-cutting,” *JM3*, vol. 14, no. 1, pp. 011 002–011 002, 2015.

[8] Y. Lin, X. Xu, B. Yu, R. Baldick, and D. Z. Pan, “Triple/quadruple patterning layout decomposition via linear programming and iterative rounding,” *JM3*, vol. 16, no. 2, 2017.

[9] X. Li, Z. Zhu, and W. Zhu, “Discrete relaxation method for triple patterning lithography layout decomposition,” *IEEE TC*, vol. 66, no. 2, pp. 285–298, 2017.

[10] S.-Y. Fang, Y.-W. Chang, and W.-Y. Chen, “A novel layout decomposition algorithm for triple patterning lithography,” *IEEE TCAD*, vol. 33, no. 3, pp. 397–408, March 2014.

[11] H.-A. Chien, S.-Y. Han, Y.-H. Chen, and T.-C. Wang, “A cell-based row-structure layout decomposer for triple patterning lithography,” in *Proc. ISPD*, 2015, pp. 67–74.

[12] H. Tian, H. Zhang, Q. Ma, Z. Xiao, and M. D. F. Wong, “A polynomial time triple patterning algorithm for cell based row-structure layout,” in *Proc. ICCAD*, 2012, pp. 57–64.

[13] J. Kuang and E. F. Y. Young, “Fixed-parameter tractable algorithms for optimal layout decomposition and beyond,” in *Proc. DAC*, 2017, pp. 61:1–61:6.

[14] ——, “An efficient layout decomposition approach for triple patterning lithography,” in *Proc. DAC*, 2013, pp. 69:1–69:6.

[15] I. H.-R. Jiang and H.-Y. Chang, “Multiple patterning layout decomposition considering complex coloring rules and density balancing,” *IEEE TCAD*, vol. 36, no. 12, pp. 2080–2092, 2017.

[16] E. Bengoetxea, “Inexact graph matching using estimation of distribution algorithms,” Ph.D. dissertation, Ecole Nationale Supérieure des Télécommunications, Paris, France, Dec 2002.

[17] W. Li, Y. Ma, Q. Sun, Y. Lin, I. H.-R. Jiang, B. Yu, and D. Z. Pan, “OpenMPL: An open source layout decomposer,” in *Proc. ASICON*, 2019.

[18] M. Schlichtkrull, T. N. Kipf, P. Bloem, R. Van Den Berg, I. Titov, and M. Welling, “Modeling relational data with graph convolutional networks,” in *European Semantic Web Conference*. Springer, 2018, pp. 593–607.

[19] D. Stolee, “Isomorph-free generation of 2-connected graphs with applications,” *arXiv preprint arXiv:1104.5261*, 2011.

[20] B. D. McKay, “Isomorph-free exhaustive generation,” *Journal of Algorithms*, vol. 26, no. 2, pp. 306–324, 1998.

[21] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, “Automatic differentiation in PyTorch,” in *NeurIPS Autodiff Workshop*, 2017.

[22] M. Wang, L. Yu, D. Zheng *et al.*, “Deep graph library: Towards efficient and scalable deep learning on graphs,” *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.